Design of loadbalancer.c

Loadbalancer.c is a program which manages load for multiple HTTP servers. It works by receiving client connections, determining which server is least utilized, and bridges the client connection to the appropriate server.

Receiving client connections

The first action my loadbalancer must take is to receive connections from clients. There can be multiple clients sending packets to the load balancer at any given time, and one client should not have to wait on the completion of another's request.

To deal with multiple clients concurrently, I decided to go with a multithreaded producer/consumer model. In an attempt to keep bugs at a minimum, I opted for a model which has a smart dispatcher which hands off client server socket pairs to dumb worker threads, via a queue.

To determine the size of my worker thread pool, I used the value option flag -N, the number of concurrent requests the load balancer can handle, with a default value of 4 worker threads.

The process for how the dispatcher threads hands work to the worker threads is as follows:

The dispatcher thread listens on its socket for new clients and once a connection is received, determine the best server to forward the client to (see next section on determining the best server). The dispatcher thread then stores the client socket and the best server socket in a queue named `queue`, which is implemented as a linked list. I decided to go with the linked list implementation because it is what made the most sense to me.

`queue` has two functions, `enqueue()` and `dequeue()`. Calling these functions is protected by a mutex lock, called `q_lock`, to keep their operations thread safe.

`enqueue()` has one more synchronization function. It calls `pthread_cond_signal()` on a condition variable named `q_cond_var`. This is to let the waiting threads know that there is work to be done in the `queue`, to avoid them simply busy waiting.

The `queue` and its functions are the only communication between the dispatcher thread and the worker threads. Once a worker thread calls `dequeue()` the only job it has is to bridge the connection between the client and server socket.

Determining the best server

In order to determine which server is the least utilized, my load balancer sends periodic GET requests to each server's healthcheck, which returns the total error and total requests each server has processed. These health probes as I will call them are sent every 5 seconds or every R requests (determined by the -R flag, default 5). I chose to send the probes every 5 seconds because I do not want the servers to dedicate much of their resources responding to healthchecks, they should reserve those resources for clients. Also, I keep track of the estimated total requests for each server locally, which is very accurate unless a server is receiving considerable traffic from outside the load balancer.

The first part of determining the best server is designing a system for the periodic healthchecks.

My main dispatcher thread hangs on `accept()` waiting for new connections, so if I handled the health probes in this thread there would be no guarantee of a probe every 5 seconds. Because of this, I chose to probe using a dedicated healthcheck thread.

Creating the healthcheck thread

After parsing the server ports from the command line, the dispatcher thread stores each port in an array called `server_ports`. This effectively labels each server by an index in the array, which is very useful for easily keeping track of which server is associated with what values.

Upon creation of the healthcheck thread, the dispatch thread passes in two arguments (in a struct of course to comply with `pthread_create`), `server_ports` and `healthcheck_vals`.

`healthcheck_vals` is an array of tuples of style (total_requests, total_errors), with the index of `healthcheck_vals` corresponding to the index of the server in server ports. Before passing it into the healthcheck thread, all values in `healthcheck_vals` are initialized to `INT_MAX`.

`INT_MAX` is my way of flagging the server as not healthy (down, unresponsive, extremely slow, etc). I initialize all the values of `healthcheck_vals` to it because my load balancer must assume all servers are unhealthy until proven healthy.

Implementing the healthcheck probe within the healthcheck thread

The function which the healthcheck thread is running communicates with the dispatch thread to decide when to send a probe, using `pthread_cond_timedwait()`. For `pthread_cond_timedwait()` to work as expected, there needs to be an associated mutex. However, in my design this mutex is not locking a critical region, it is just there for `pthread_cond_timedwait()` to work properly, so I do not feel the need to describe it further.

At the very beginning of my infinite loop within the healthcheck thread, I call `pthread_cond_timedwait()`. This thread now waits either 5 seconds, or until it is signaled using `pthread_cond_signal()` by the dispatcher. The dispatcher keeps track of the total requests sent in by clients and calls `pthread_cond_signal()` when the total number of requests % the r value == 0.

Once the healthcheck thread is done waiting or is signaled, it calles `probe_server()`.

`probe_server()` takes in `server_ports` as its only argument. It works by using a for loop to loop through all the indices in `server_ports`, and at each index, creating a socket to the server using the port number and sending a healthcheck request. `probe_server()` then reads the servers response from the socket and parses the data. Next, it updates the `healthcheck_vals` for the server with the total requests and errors it parsed, which is a critical region and requires locking. After updating, `probe_server()` closes the server socket.

After doing this process for all servers, `healthcheck_vals` holds the latest information on the health of each server.

How the dispatch thread uses server health information
Back in the dispatch thread, after accepting a new client connection, the thread calls grabs the associated mutex for `healthcheck_vals` and calls `determine_best_server()`. It needs to have the `healthcheck_vals` mutex since the entirety of `determine_best_server()` reads from `healthcheck_vals` and is thus a critical region.

`determine_best_server()` only takes `healthcheck_vals` as an argument and returns the index of the least utilized server. The logic is as follows:

```
determine_best_server(healthcheck_vals)
     best_server = index of min(healthcheck_vals.total_req)
     If there is a tie, best_server = index of min(errors/total_req)
     If there is still a tie, best_server = random contending index
     return best_server
```

Once `determine_best_server()` returns, the dispatch thread releases the mutex for `healthcheck_vals`. Next the dispatcher thread creates a socket to the best server determined and enques the client socket, server socket pair for the worker threads.

Bridging the connections

After dequeuing, the worker threads bridge the connection between client and server using `bridge_loop()` and `bridge_connection()`, which were provided in the started code, allowing the client and server to carry out their communication.

The worker threads bridge the connection until one party closes the connection. The workers then close both sockets and continue to look for work on the queue.

I purposely made the worker threads extremely simple to make them act as a black box, allowing me to only have to worry about the more difficult part of the program, determining the best server.

Errors

My load balancer detects and handles errors using the healthcheck and dispatcher threads. The healthcheck thread communicates to the dispatcher which servers are unhealthy by setting the servers `healthcheck_vals` to `INT_MAX`. The healthcheck thread will mark a server unhealthy for this list of reasons:

- The server failed to connect
- The server did not respond to the healthcheck request within 1 second
- The server responded with a status code that is not 200
- The server responded with a malformed content

Before enquing a connection, the dispatcher thread checks if the best servers `healthcheck_vals` are `INT_MAX`. If they are, the dispatcher knows that server is experiencing an error and sends a status code 500 Internal Server Error to the client and closes the connection.